Remarks

Claims 1-36 are pending in the application. Claim 21 was rejected under 35 U.S.C. § 102(a) in view of the article Richard Grimes, "Attribute Programming with Visual C++" ["Grimes article"]. Claims 1-20 were rejected under 35 U.S.C. § 103(a) as being unpatentable over the Grimes article in view of selected portions of the book Aho et al., Compilers Principles, Techniques, and Tools ["Aho book"].

Applicants have added dependent claims 22-36.

Applicants respectfully request reconsideration of the application in view of the foregoing amendments and following remarks.

I. The Grimes article

With the goal of establishing a shared understanding of the disclosure of the Grimes article, Applicants respectfully make the following observations.

Microsoft provided certain technical preview files at a Microsoft Professional Developer's Conference in Denver in 1998. The Grimes article describes attribute programming with Visual C++, as that topic was understood by the author Richard Grimes after his analysis of the technical preview files. Grimes notes that the technical preview files supplied "an attribute programming-aware compiler and an example of how to use it. There is no documentation, other than how to set up the compiler and build the single example." [Grimes article, page 1.] Grimes then hypothesizes about the capabilities of the compiler – "I will take this opportunity to give you some pointers as to what you can do with this preview and to determine what attribute programming will be like in the future." [*Id.*] Grimes also provides the following disclaimer:

> The information I present here has been obtained by looking into the compiler DLLs to get an idea of the attributes that you can use and then determining by trial and error how to use them. Therefore I may not have found the easiest, or even the correct, way to perform these actions. [*Id.*]

After laying that foundation, Grimes gives general examples of attribute programming with attributes such as "comobject." Grimes generally describes, with reference to a figure on page 3, interaction between a compiler and an attribute provider, in which the "Compiler passes the Attribute and its parameters to provider" and the "Provider tells compiler how to change the code." [Grimes article, figure on page 3.] Back in the compiler, the "combination of the original and the 'injected' code is compiled to generate an .obj file." [Grimes article, page 3.]

Most of the Grimes article is about attribute programming generally. Grimes briefly addresses interface definition, however. Specifically, Grimes writes:

> The technical preview integrates interface (IDL) definition into the C++ code that uses it. To do this Microsoft has added the 'interface' keyword to the C++ language. *This acts in the same way as the same keyword in IDL*: it describes an interface that defines how the client and object will talk to each other. [Grimes article, page 2 (emphasis added).]

As indicated by Grimes, the use of IDL keywords and IDL attributes in the technical preview materials was limited; they acted "in the same way as" in IDL. As repeatedly described by Grimes, the only use of IDL keywords and IDL attributes in C++ code in the technical preview was to generate IDL in the .obj file.

> Since interface definition is so important, one would hope that Microsoft will give developers the choice of whether to use MIDL or not, and this appears to be the purpose of the [emitidl] attribute that is used in the preview example. This indicates that IDL should be generated from the attributes and placed into the compiled .obj file. The preview comes with a tool called Idlgen that appears to do the dual step of generating an IDL file from the .obj file and then running it through MIDL to generate a type library and the standard marshaling (proxy-stub) files. [Grimes article, page 2.]
>
> . . .
>
> Some attributes, however, do not generate C++ code, instead, they indicate that the object will use COM+ Services (like transactions). The compiler handles this by storing this information as a custom attribute in the project's type library. The compiler also adds the attribute information to the .obj file which can be extracted with the Idlgen utility to generate IDL and then runs MIDL to create the type information. [Grimes article, page 3.]
>
> . . .

This is the purpose of the emitidl attribute, but as I mentioned earlier, this places IDL into the .obj file, so you need to run the IDLGEN utility to emit this data as an IDL file and to run MIDL to compile the IDL. You can call this tool as a post-build step in the project settings. [Grimes article, page 5.]

Thus, in the Grimes article, Grimes describes (1) attribute programming generally, and (2) generating IDL from IDL attributes and placing the generated IDL into an .obj file (and then optionally using the IDLGEN utility on the .obj file).

II. <u>Rejection of claims 1-20 in view of the Grimes article and the Aho book</u>

Claims 1-20 were rejected as being unpatentable over the Grimes article in view of the Aho book. Applicants respectfully disagree. The Grimes article and Aho book, taken separately or in combination, fail to teach or suggest at least one limitation from each of claims 1-20.

A. Claims 1-5, 10-15, 18, and 19

Claim 1, as amended, recites:

a back end module that produces output <u>computer-executable</u> code from the intermediate representation based at least in part upon semantics of the embedded definition language information.

According to claim 1, as amended, a compiler system performs semantic analysis of definition language information embedded in programming language code in a file. The compiler system includes a front end module, a converter module, and a back end module. The front end module separates a file into plural tokens, where the file includes programming language code having embedded therein definition language information. The converter module converts the plural tokens into an intermediate representation. The back end module produces output <u>computer-executable</u> code from the intermediate representation based at least in part upon semantics of the embedded definition language information. For example, in one embodiment of the invention of claim 1:

The C++ compiler environment derives semantic meaning from the IDL embedded in C++ code, which enables powerful interface-based programming capabilities. For example, the C++ compiler can automatically generate an implementation for a dispatch interface using an extremely clean and sparse syntax. The C++ compiler can also automatically generate code for a client-side dispatch interface call site, which greatly simplifies programming. [Application at page 9, lines 18-23.]

Overall, claims 10, 13, and 18 each recites different language than claim 1, but each includes language similar to the above-cited language from claim 1. Specifically, claim 10, as amended, recites:

<u>wherein the one or more output code files further include output computer-executable code based at least in part upon semantics of the definition language constructs</u>

Claim 13, as amended, recites:

generating output <u>computer-executable</u> code from the intermediate representation based at least in part upon semantics of the embedded definition language information.

And, claim 18, as amended, recites:

a back end module that produces output <u>computer-executable</u> code from the intermediate representation based at least in part upon semantics of the definition language information.

The Grimes article and the Aho book, taken separately or in combination, fail to teach or suggest the above-cited language from each of claims 1, 10, 13, and 18, respectively.

Suppose for the sake of argument that the Grimes article describes a file that includes programming language having embedded or mixed therein definition language. Even so, the Grimes article does not teach or suggest the above-cited language from each of claims 1, 10, 13, and 18, respectively. In its general discussion of attribute programming, the Grimes article describes injecting C++ code in response to certain kinds of attributes (other than IDL attributes). [Grimes article, page 3.] For IDL attributes, however, the Grimes article describes generating IDL from the attributes and placing the generated IDL into an .obj file [Grimes article, pages 2, 3, and 5], which does not involve output computer-executable code based upon the semantics of the definition language information or constructs. In fact, the Grimes article leads away from the above-cited language of each of claims 1, 10, 13, and 18, respectively. When Grimes writes, "Defining interfaces like this within C++ is a little restrictive" [Grimes at page 2], Grimes suggests that mixing definition language and programming language is undesirable.

The Aho book also does not teach or suggest the above-cited language from each of claims 1, 10, 13, and 18, respectively. The Aho book generally describes compiler techniques and tools. The parts of the Aho book specifically cited by the Examiner describe, among other things, compiler organization by "front end" and "back end" [Aho book, page 20], symbol tables [Aho book, page 11], and parse trees [Aho book, pages 6 and 40-48], but do not address working with both programming language and definition language in one file.

The Grimes article and the Aho book separately fail to teach or suggest the above-cited language from each of claims 1, 10, 13, and 18, respectively. Accordingly, the combination of the Grimes article and the Aho book also fails to teach or suggest the language.

Claims 1, 10, 13, and 18 should be allowable. In view of the foregoing discussion of claims 1, 10, 13, and 18, Applicants will not belabor the merits of the separate patentability of dependent claims 2-5, 11, 12, 14, 15, and 19. Claims 1-5, 10-15, 18, and 19 should be allowable.

B. Claims 6-9, 16, 17, and 20

Claim 6 recites:

wherein the symbol table includes plural entries for symbol names for the programming language constructs, at least one of the plural entries having an associated list of interface definition language attributes, and wherein the parse tree unifies representation of the programming language constructs and the interface definition language constructs.

Claim 9 recites:

at least one of the plural entries having an associated list of interface definition language attributes based upon the embedded interface definition language

information; and
a second data field storing data representing a parse tree, wherein the parse tree unifies representation of the programming language code and the embedded interface definition language information.

Claim 16, as amended, recites:

building a symbol table having plural entries for symbol names for the programming language code, at least one of the plural entries having an associated list of definition language attributes based upon the embedded definition language information; and building a ~~parse~~ tree that unifies representation of the embedded definition language information and the programming language code.

And, claim 20, as amended, recites:

modifying the programming language compiler to allow manipulation of the symbol table and the parse tree by the one or more interface definition language attribute providers based upon the semantics of the embedded interface definition language information.

The Grimes article and the Aho book, taken separately or in combination, fail to teach or suggest the above-cited language from each of claims 6, 9, 16, and 20, respectively.

Suppose for the sake of argument that the Grimes article describes a file that includes programming language having embedded or mixed therein definition language. Even so, the Grimes article does not teach or suggest the above-cited language from each of claims 6, 9, 16, and 20, respectively. In its general discussion of attribute programming, the Grimes article describes injecting C++ code in response to certain kinds of attributes (other than IDL attributes). [Grimes article, page 3.] For IDL attributes, however, the Grimes article describes generating IDL from the attributes and placing the IDL into an .obj file [Grimes article, pages 2, 3, and 5], which leads away from the above-cited language from each of claims 6, 9, 16, and 20, respectively. The Grimes article further leads away from the above-cited language of each of claims 6, 9, 16, and 20, respectively, when Grimes writes, "Defining interfaces like this within C++ is a little restrictive" [Grimes at page 2], which suggests that mixing definition language and programming language is undesirable.

The Aho book also does not teach or suggest the above-cited language from each of claims 6, 9, 16, and 20, respectively. The Aho book generally describes compiler techniques and tools. The parts of the Aho book specifically cited by the Examiner describe, among other things, compiler organization by "front end" and "back end" [Aho book, page 20], symbol tables [Aho book, page 11], and parse trees [Aho book, pages 6 and 40-48], but do not address working with both programming language and definition language.

The Grimes article and the Aho book separately fail to teach or suggest the above-cited language from each of claims 6, 9, 16, and 20, respectively. Accordingly, the combination of the Grimes article and the Aho book also fails to teach or suggest the language.

Claims 6, 9, 16, and 20 should be allowable. In view of the foregoing discussion of claims 6, 9, 16, and 20, Applicants will not belabor the merits of the separate patentability of dependent claims 7, 8, and 17. Claims 6-9, 16, 17, and 20 should be allowable.

III. Rejection of claim 21 in view of the Grimes article

Claim 21 recites:

> embedding by the programming language compiler debugging information in a definition language output file, the definition language output file for subsequent processing by a definition language compiler, whereby the embedded debugging information associates errors raised by the definition language compiler with locations of embedded definition language constructs in the input file to facilitate debugging of the input file.

The Grimes article fails to teach or suggest the above-cited language from claim 21. Suppose for the sake of argument that the Grimes article describes: (1) generating IDL from IDL attributes and placing the generated IDL into an .obj file [Grimes article, page 2]; and (2) using a tool called IDLGEN to generate an IDL file from the .obj file then run the IDL file through MIDL [*Id.*]. Even so, generating an IDL file and running the IDL file through MIDL does not teach or suggest *embedding debugging information in* a definition language output file.

Apart from its discussion of IDL files and the IDLGEN tool, the Grimes article generally discusses attribute programming. The Grimes article describes analysis of debug builds in which a compiler stores (in a project's PDB file) information about ATL code that was generated. [Grimes article at 3.] A PDB file is a binary file that contains debugging information gathered over the course of compiling and linking the project, and is for use with the Visual C++ debugger. The Grimes article's description of PDB files relates to debug files for injected programming language code, not embedding debugging information in a definition language output file. This portion of the Grimes article also does not teach or suggest the above-cited language of claim 21.

Claim 21 should be allowable.

IV. Notes from the Examiner

The Examiner ends the Office action with two "special notes" in which the Examiner sketches out alternative grounds of rejection for claims 1-21. In these notes, the Examiner has not provided specific references in support the rejections of claims 1-21, nor has the Examiner specifically mapped claim language to any reference. Nevertheless, Applicants will attempt to address the merits of the notes. Should the Examiner persist in maintaining any rejection described in the notes section, Applicants respectfully request that the Examiner provide specific citations to references and specific mappings to the claim language.

In both of the notes, the Examiner addresses interpretation of the term "definition language information." Applicants respectfully note that not all of claims 1-21 recite the term "definition language information." [*See*, *e.g.*, claims 6-8.] Further, several claims recite the term "definition language information" but recite an additional "definition language" term as well. [*See*, *e.g.*, claim 10.]

In the first note, the Examiner writes:

Claims 1-21 could also be rejected with single reference Aho in that the phrase "definition language information" could broadly be interpreted as an abstract class. Under this definition, an abstract class defines the look of an upcoming concrete class. Clearly, one compiler will interpret and compile object code for both the "definition language information" (abstract class) and the programming language (concrete class).

Applicants fail to understand the Examiner's example. Conventionally, an abstract class and a concrete class are coded using the same constructs from a single language, for example, C++.

In the second note, the Examiner writes:

Claims 1-21 could also be rejected by interpreting "definition language information" as the stub or skeletal code produced by a standard IDL compiler. Under this definition a single compiler will interpret and compile the stub code along with the programming code (implementation written by programmer) as a normal operation. Key to this interpretation is the broad meaning of the word "information", which could be anything and any step in the processing of IDL.

Applicants again fail to understand the Examiner's example. The Examiner indicates that "definition language information" is the output produced by a standard IDL compiler. In contrast, it is Applicants' understanding that, for example, a MIDL compiler typically accepts IDL *as input* and generates stub code. Moreover, if it is the Examiner's position that the stub code itself is the "definition language information," it is Applicants' understanding that stub code is conventionally executable code and is not mixed or embedded in a given file with programming language code.

V. Dependent claims 22-36

Applicants respectfully add claims dependent claims 22-36. In view of the foregoing discussion of the independent claims from which claims 22-36 depend, the merits of the separate patentability of claims 22-36 are not belabored at this point. Claims 22-36 should be allowable.

Claims 22, 23, 27, 28, 32, and 33 are supported in the application as filed, for example, at pages 30-31. Claims 24, 25, 29, 30, 34, and 35 are supported in the application as filed, for example, at page 22. Claims 26, 31, and 36 are supported in the application as filed, for example, at pages 9-10.

Conclusion

The claims in their present form should now be allowable. Such action is respectfully requested.

Respectfully submitted,

KLARQUIST SPARKMAN, LLP

By [illegible]
Kyle B. Rinehart
Registration No. 47,027

One World Trade Center, Suite 1600
121 S.W. Salmon Street
Portland, Oregon 97204
Telephone: (503) 226-7391
Facsimile: (503) 228-9446

(147268.1)